2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Short-Circuiting Memory Traffic in Handheld Platforms

Praveen Yedlapalli* Nachiappan Chidambaram Nachiappan* Niranjan Soundararajan†
Anand Sivasubramaniam* Mahmut T. Kandemir* Chita R. Das*
* *The Pennsylvania State University* † *Intel Corp.*
*Email: *{praveen, nachi, anand, kandemir, das}@cse.psu.edu, †{niranjan.k.soundararajan}@intel.com,*

***Abstract*—Handheld devices are ubiquitous in today's world. With their advent, we also see a tremendous increase in device-user interactivity and real-time data processing needs. Media (audio/video/camera) and gaming use-cases are gaining substantial user attention and are defining product successes. The combination of increasing demand from these use-cases and having to run them at low power (from a battery) means that architects have to carefully study the applications and optimize the hardware and software stack together to gain significant optimizations. In this work, we study workloads from these domains and identify the memory subsystem (system agent) to be a critical bottleneck to performance scaling. We characterize the lifetime of the "frame-based" data used in these workloads through the system and show that, by communicating at frame granularity, we miss significant performance optimization opportunities, caused by large IP-to-IP data reuse distances. By carefully breaking these frames into sub-frames, while maintaining correctness, we demonstrate substantial gains with limited hardware requirements. Specifically, we evaluate two techniques, flow-buffering and IP-IP short-circuiting, and show that these techniques bring both power-performance benefits and enhanced user experience.**

I. Introduction

The propensity of tablets and mobile phones in this handheld era raises several interesting challenges. At the application end, these devices are being used for a number of very demanding scenarios (unlike the mobiles of a decade ago), requiring substantial computational resources for real-time interactivity, on both input and output sides, with the external world. On the hardware end, power and limited battery capacities mandate high degrees of energy efficiencies to perform these computational tasks. Meeting these computational needs with the continuing improvements in hardware capabilities is no longer just a matter of throwing high performance and plentiful cores or even accelerators at the problem. Instead, a careful examination and marriage of the hardware with the application and execution characteristics is warranted for extracting the maximum efficiencies. In other words, a co-design of software and hardware is necessary to design energy- and performance-optimal systems, which may not be possible just by optimizing the system in parts. With this philosophy, this paper focusses on an important class of applications run on these handheld devices (real-time frame-oriented video/graphics/audio), examines the data flow in these applications through the different computational kernels, identifies the inefficiencies when sustaining these flows in today's hardware solutions, which simply rely on main memory to exchange such data, and proposes alternate hardware enhancements to optimize such flows.

Real-time interactive applications, including interactive games, video streaming, camera capture, and audio playback, are amongst the most popular on today's tablets and mobiles apart from email and social networking. Such applications account for nearly 65% of the usage on today's handhelds [1], stressing the importance of meeting the challenges imposed by such applications efficiently. This important class of applications has several key characteristics that are relevant to this study. First, these applications work with input (sensors, network, camera, etc.) and/or output (display, speaker, etc.) devices, mandating real-time responsiveness. Second, these applications deal with "frames" of data, with the requirement to process a frame within a stipulated time constraint. Third, the computation required for processing a frame can be quite demanding, with hardware accelerators[1] often deployed to leverage the specificity in computation for each frame and delivering high energy efficiency for the required computation. The frames are then pipelined through these accelerators one after another sequentially. Fourth, in many of these applications, the frames have to *flow* not just through one such computational stage (accelerator) but possibly though several such stages. For instance, consider a video capture application, where the camera IP may capture raw data, which is then encoded into an appropriate form by another IP, before being sent either to a flash storage or a display. Consequently, the frames have to flow through all these computational stages, and typically the memory system (the DRAM main memory) is employed to facilitate this flow. Finally, we may need to support several such flows at the same time. Even a single application may have several concurrent flows (the video part and audio part of the video capture application which have their own pipelines). Even otherwise, with multiprogramming increasingly prevalent in handhelds, there is a need to concurrently support individual application flows in such environments.

The authors would like to confirm that this work is an academic exploration and does not reflect any effort within Intel.

[1]We use the term accelerators and IPs interchangeably in this work.

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.60

Apart from the computational needs for real-time execution, all the above observations stress the memory intensity of these applications. Frames of data coming from any external sensor/device is streamed in to memory, from which it is streamed out by a different IP, processed and put back in memory. Such an undertaking places heavy demands on the memory subsystem. When we have several concurrent flows, either within the same application or across applications in a multiprogrammed environment, all of these flows contend for the memory and stresses it even further. This contention can have several consequences: (i) without a steady stream of data to/from memory, the efficiencies from having specialized IPs with continuous dataflow can get lost with the IPs stalling for memory; (ii) such stalls with idle IPs can lead to energy wastage in the IPs themselves; and (iii) the high memory traffic can also contend with, and slow down, the memory accesses of the main cores in the system. While there has been a lot of work covering processing – whether it be CPU cores or specialized IPs and accelerators (e.g. [35] [50] [27]) – for these handheld environments, the topic of optimizing the data flows, while keeping the memory system in mind, has drawn little attention. Optimizing for memory system performance, and minimizing consequent queueing delays has itself received substantial interest in the past decade, but only in the area of high-end systems (e.g., [4] [26] [25] [10]). This paper addresses this critical issue in the design of handhelds, where memory will play an increasingly important role in sustaining the data flow not just across CPU cores, but also between IPs, and with the peripheral input-output (display, sound, network and sensors) devices.

In today's handheld architectures, a *System Agent* (SA) [8], [23], [31], [48] serves as the glue integrating all the compute (whether it be IPs or CPU cores) and storage components. It also serves as the conduit to the memory system. However, it does not clearly understand data flows, and simply acts as a slave initiating and serving memory requests regardless of which component requests it. As a result, the high frame rate requirements translate to several transactions in the memory queues, and the flow of these frames from one IP to another explicitly goes through these queues, i.e., the potential for data flow (or data *reuse*) across IPs is not really being exploited. Instead, in this paper we explore the idea of *virtually integrating accelerator pipelines* by "short-circuiting" many of the read/write requests, so that the traffic in the memory queues can be substantially reduced. Specifically, we explore the possibility of *shared buffers/caches* and *short-circuiting communication* between the IP cores based on requests already pending in the memory transaction queues. In this context, this paper makes the following **contributions**:

- We show that the memory is highly utilized in these systems, with IPs facing around 47% of their total execution time stalling for data, in turn, causing 24% of the frames to be dropped in these applications. We cannot afford to let technology take care of this problem since with each DRAM technology advancement, the demands from the memory system also become more stringent.
- Blindly provisioning a shared cache to leverage data flow/reuse across the IP cores is also likely to be less beneficial from a practical standpoint. An analysis of the IP-to-IP reuse distances suggests that such caches have to run into several megabytes for reasonable hit rates (which would also be undesirable for power).
- We show that this problem is mainly due to the current frame sizes being relatively large. Akin to tiling for locality enhancement in nested-loops of large arrays [28], [43], [49], we introduce the notion of "*sub-frame*" for restructuring the data flow, which can substantially reduce reuse distances. We discuss how this sub-framing can be done for the considered applications.
- With this sub-framing in place, we show that reasonably sized shared caches – referred to as *flow buffers* in this paper – between the producer and consumer IPs of a frame can circumvent the need to go to main memory for many of the loads from the consumer IP. Such reduction in memory traffic results in around 20% performance improvement in these applications.
- While these flow buffers can benefit these platforms substantially, we also explore an alternate idea of not requiring any separate hardware structures – leveraging existing memory queues for data forwarding from the producer to the consumer. Since memory traffic is usually high, recently produced items are more likely to be waiting in these queues (serving as a small cache), which could be forwarded to the requesting consumer IP. We show that this can be accommodated in recently-proposed memory queue structures [5], and demonstrate performance and power benefits that are nearly as good as that of the flow buffer solution.

II. Background and Experimental Setup

In this section, we first provide a brief overview of current SoC (system-on-chip) platforms showing how the OS, core and the IPs interact from a software and hardware perspective. Next, we describe our evaluation platform, and properties of the applications that are used in this work.

A. Overview of SoC Platforms

As shown in Figure 1, handhelds available in the market have multiple cores and other specialized IPs. The IPs in these platforms can be broadly classified into two categories – *accelerators* and *devices*. Devices interact directly with the user or external world and include cameras, touch screen, speaker and wireless. Accelerators are the on-chip hardware components which specialize in certain activities. They are

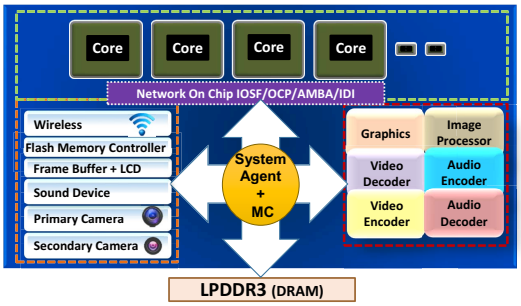

Figure 1: Target SoC platform with a high-level view of different functional blocks in the system.

the workhorses of the SoC as they provide maximum performance and power efficiency, e.g. video encoders/decoders, graphics, imaging and audio engines.

Interactions between Core, IPs and Operating System: SoC applications are highly interactive and involve multiple accelerators and devices to enhance user experience. Using API calls, application requirements get transformed to accelerator requirements through different layers of the OS. Typically, the calls happen through software device drivers in the kernel portion of the OS. These calls decide if, when and for how long the different accelerators get used. The device drivers, which are optimized by the IP vendors, control the functionality and the power states of the accelerators. Once an accelerator needs to be invoked, its device driver is notified with request and associated physical address of input data. The device driver sets up the different activities that the accelerator needs to do, including writing appropriate registers with pointers to the memory region where the data should be fetched and written back. The accelerator reads the data from main memory through DMA. Input data fetching and processing are pipelined and the fetching granularity depends on how the local buffer is designed. Once data is processed, it is written back to the local buffers and eventually to the main memory at the address region specified by the driver. As most accelerators work faster than main memory, there is a need for input and output buffers.

The System Agent (SA): Also known as the Northbridge, is a controller that receives commands from the core and passes them on to the IPs. Some designs add more intelligence to the SA to prioritize and reorder requests to meet QoS deadlines and to improve DRAM hits. SA usually incorporates the memory controller (MC) as well. Apart from re-ordering requests across components to meet QoS guarantees, even fine-grained re-ordering among IP's requests can be done to maximize DRAM bandwidth and bus-utilization. With increasing user demands from handhelds the number of accelerators and their speeds keep increasing [12], [13], [44]. These trends will place a very high demand on DRAM traffic. Consequently, unless we design a sophisticated SA that can handle the increased amount of traffic, the improvement in accelerators' performance will not end in improved user experience.

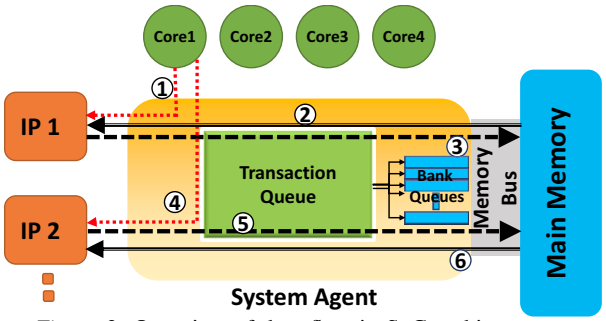

Figure 2: Overview of data flow in SoC architectures.

B. Data movement in SoCs

Figure 2 depicts the high-level view of the data flow in SoC architectures. Once a core issues a request to an IP through he SA (shown as (1), the IP starts its work by injecting a memory request into SA. First, the request traverses through an interconnect which is typically a bus or cross-bar, and is enqueued in a memory transaction queue. Here, requests can be reordered by the SA according to individual IP priorities to help requests meet their deadlines. Subsequently, requests are placed in the bank-queues of the memory controller, where requests from IPs are rearranged to maximize the bus utilization (and in turn, the DRAM bandwidth). Following that, an off-chip DRAM access is made. The response is returned to the IP through the response network in the SA (shown as (2)). IP-1 writes its output data to memory (shown in (3)) till it completes processing the whole frame. After IP-1 completes processing, IP-2 is invoked by the core (shown as 4), and data flow similar to what IP-1 had is followed, as captured by (5) and (6) in Figure 2. The unit of data processing in media and gaming IPs (including audio, video and graphics) is a **frame**, which carries information about the image or pixels or audio delivered to the user. *Typically a high frame drop rate corresponds to a deterioration in user-experience.*

The total request access latency includes the network latency, the queuing latencies at the transaction queue and bank queue, DRAM service time, and the response network latency. This latency, as expected, is not constant and varies based on the system dynamics (including DRAM row-buffer hits/misses). When running a particular application, the OS maps the data frames of different IPs to different physical memory regions. Note that these regions get reused during the application run for writing different frames (over time). In a data flow involving multiple IPs that process the frames, one after another, the OS (through device drivers) synchronizes the IPs such that the producer IP writes another frame of data onto the same memory region after the consumer IP had consumed the earlier frame.

C. Decomposing an Application Execution into Flows

Applications cannot directly access the hardware for I/O or acceleration purposes. In Android, for example, application requests get translated by intermediate libraries and

IP Abbr.	Expansion	IP Abbr.	Expansion
VD	Video Decoder	AD	Audio Decoder
DC	Display Controller	VE	Video Encoder
MMC	Flash Controller	MIC	Microphone
AE	Audio Encoder	CAM	Camera
IMG	Imaging	SND	Sound

Table I: Expansions for IP abbreviations used in this paper.

Id	Application	IP Flows
A1	Angry Birds	AD - SND; GPU - DC
A2	Sound Record	MIC - AE - MMC
A3	Audio Playback (MP3)	MMC - AD - SND
A4	Photos Gallery	MMC - IMG - DC
A5	Photo Capture (Cam Pic)	CAM - IMG - DC; CAM - IMG - MMC
A6	Skype	CAM - VE; VD - DC; AD - SND; MIC - AE
A7	Video Record	CAM - VE - MMC; MIC - AE - MMC
A8	Youtube	VD - DC; AD - SND

Table II: IP flows in our applications.

device drivers into commands that drive the accelerators and devices. This translation results in hardware processing the data, moving it between multiple IPs and finally writing to storage or displaying or sending it over the network. Let us consider for example a video player application. The flash controller reads a chunk of video file from memory, gets processed by the core, and two separate requests are sent to video-decoder and audio-decoder. They read their data from the memory and, once an audio/video frame is decoded, it is sent to the display through memory. In this paper, we term such a regular stream of data movement from one IP to another as a **flow**. All our target applications have such flows, as shown in Table II. Table I gives the expansions for IP abbreviations. It is to be noted that an application can have one or more flows. In multiple flows, each flow could be invoked at a different point in time, or multiple independent flows can be active concurrently *without* sharing any common IP or data across them.

D. Evaluation Platform

Handheld/mobile platforms commonly run applications that rely on user inputs and are interactive in nature. Studying such a system is tricky due to the non-determinism associated with it. To enable that, we use GemDroid [7], which utilizes Google Android's open-source emulator [16] to capture the complete system-level activity. This provides a complete memory trace (with cycles between memory accesses) along with all IP calls when they were invoked by the application. We extended the platform by including DRAMSim2 [34] for accurate memory performance evaluation. Further, we enhanced the tool to extensively model the system agent, accelerators and devices in detail.

Specifications of select IPs frequency, frame sizes and processing latency are available from [42]. For completeness, we give all core parameters, DRAM parameters, and IP details in Table III. The specifications used are derived from current systems in the market [3], [36]. Note that the techniques discussed in this work are generic and not tied to specific micro-architectural parameters.

Processor	ARM ISA; 4-core processor; Clocked at 2 GHz; OoO w/issue width: 4
Caches	32 KB L1-I; 32KB L1-D; 512 KB L2
Memory	Till 2 GB reserved for cores. 2GB to 3GB reserved for IPs. LPDDR3-1333; 1 channel; 1 rank; 8 Banks 5.3 GBPS peak bandwidth; t_{CL},t_{RP},t_{RCD} = 12, 12, 12 ns
System Agent	Frequency: 500 MHz; Interconnect latency: 1 cycle per 16 Bytes Memory Transaction-Q.: 64 entries; Bank-Q.: 8 entries
IPs and System Parameters	All IPs run at 500Mhz frequency Aud.Frame: 16KB frame; Vid.Frame: 4K (3840x2160) Camera Frame: 1080p (1920x1080) Input Buffer Sizes: 16-32KB; Output Buffer Sizes: 32-64KB Enc/Decoding Ratio: VD→1:32; AD→1:8;

Table III: Platform configuration.

III. Motivation: Memory Stalls

In this section, we show that DRAM stalls are high in current SoCs and this will only worsen as IPs performance scale. Typically, DRAM is shared between the cores and IPs and is used to transfer data between them. There is a high degree of data movement and this often results in a high contention for memory controller bandwidth between the different IPs [21]. Figure 3 shows the *memory bandwidth obtained* by two of our applications: `YouTube` and `Skype` with a 6.4 GBPS memory. One can notice the *burstiness* of the accesses in these plots. Depending on the type of IPs involved, frames get written to memory or read from memory at a certain rate. For example, cameras today can capture video frames of resolution 1920x1080 at 60 FPS and the display refreshes the screen with these frames at the same rate (60 FPS). Therefore, 60 bursts of memory requests from both IPs happen in a second, with each burst requesting one complete frame. While the request rate is small, the data size per request is high – 6MB for a 1920x1080 resolution frame (this will increase with 4K resolutions [12]). If this amount of bandwidth cannot be catered to by the DRAM, the memory controller and DRAM queues fill up rapidly and in turn the devices and accelerators start experiencing performance drops. The performance drop also affects battery life as execution time increases. In the right side graph in Figure 3, whenever the camera (CAM) initiates its burst of requests, the peak memory bandwidth consumption can be seen (about 6 GBPS). We also noticed that the average memory latency more than doubles in those periods, and memory queues sustain over 95% utilization.

To explain how much impact the memory subsystem and the system-agent can have on IPs' execution time (*active cycles* during which the IPs remains in active state), in Figure 4, we plot the total number of cycles spent by an IP in processing data and in data stalls. Here, we use "data stall"

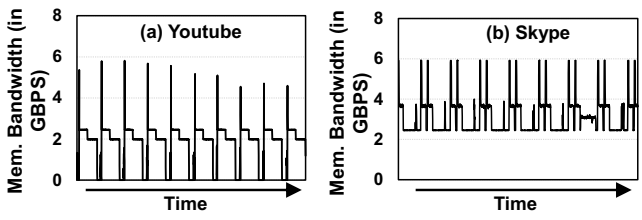

Figure 3: Bandwidth usage of `Youtube` and `Skype` over time.

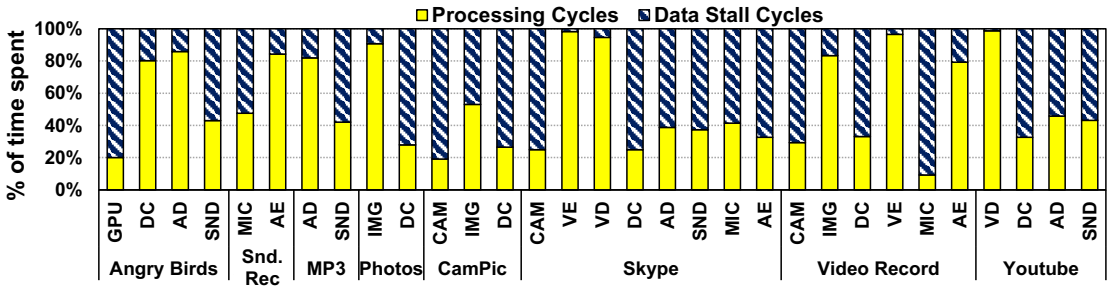

Figure 4: Total data stalls and processing time in IPs during execution.

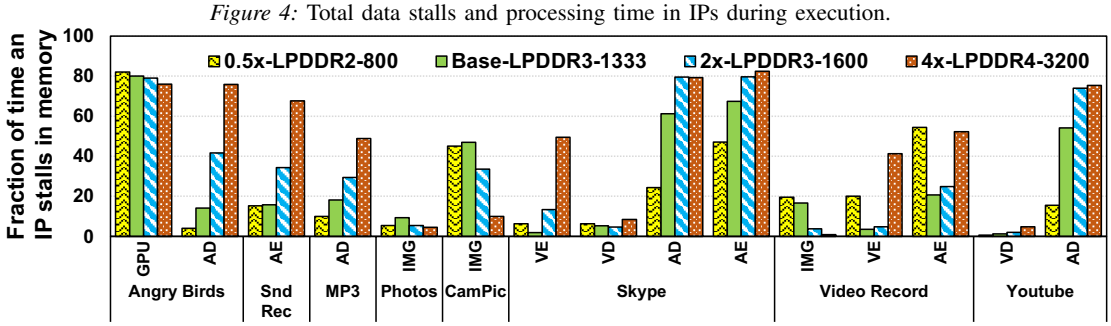

Figure 5: Trends showing increase of percentage of data stalls with each newer generation of IPs and DRAMs.

to mean the number of cycles an IP stalls for data without doing any useful computation, after issuing a request to the memory. We observe from Figure 4 that the video decoder and video encoder IPs spend most of their time processing the data, and do not stress the memory subsystem. IPs that have very small compute time, like the audio decoder and sound engine, demand very high bandwidth than what memory can provide, and thus tend to stall more than compute. Camera IP and graphics IP, on the other hand, send bursts of requests for large frames of data at regular intervals. Here as well, if memory is not able to meet the high bandwidth or has high latency, the IP remains in the high-power mode stalling for the requests. The high data stalls seen in Figure 4 translate to frame drops which is shown in Figure 6 (for 5.3 GBPS memory bandwidth). We see that on average 24% of the frames are dropped with the default baseline system, which can hurt user experience with the device. With higher memory bandwidths (2x and 4x of the baseline bandwidth), though the frame drops decrease, they still do not improve as much as the increase in bandwidth. Even with 4x baseline bandwidth, we observe more than 10% frame drops (because of higher memory latencies).

As user demands increase and more use-cases need to be supported, the number of IPs in the SoC is likely to increase [44] along with data sizes [12]. Even as the DRAM speeds increase, the need to go off-chip for data accesses places a significant bottleneck. This affects performance, power and eventually the overall user experience. To understand the severity of this problem, we conduct a simple experiment shown in Figure 5, demonstrating how the cycles per frame vary across the base system (given in Table III) and when the IPs compute at half their base speed (last generation IPs), and twice their speed (next generation) etc. For DRAM, we varied the memory throughput by varying the LPDDR configurations. We observe from the results in Figure 5 that the percentage of data stalls increases as we go from one generation to the next. Increasing the DRAM peak bandwidth alone is *not* sufficient to match the IP scaling. We require solutions that can tackle this problem within the SoC.

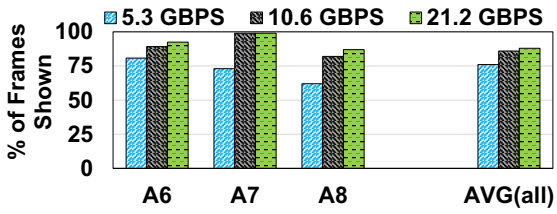

Figure 6: Percentage of frames completed in a subset of applications with varying memory bandwidths.

Further, to establish the maximum gains that can be obtained if we had an "ideal and perfect memory", we did a hypothetical study of *perfect memory with 1 cycle latency*. The cycles-per-frame results with this perfect memory system are shown in Figure 7. As expected, we observed

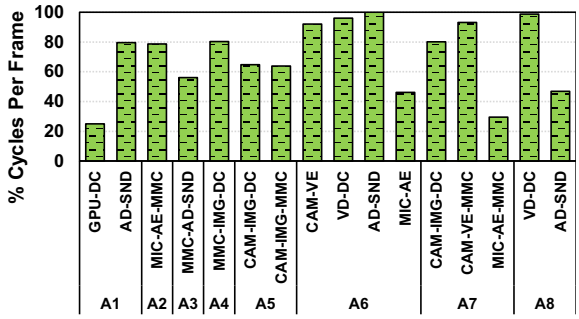

Figure 7: Percentage reduction in Cycles-Per-Frame in different flows with a perfect memory configuration.

drastic reduction in cycles per frames across applications and IPs (as high as 75%). In some IPs, memory is not a bottleneck and those did not show improved benefits. From this data, we conclude that reducing the memory access times does bring the cycles per frame down, which in turn boosts the overall application performance. Note that, this perfect memory does not allow any frames to be dropped.

IV. IP-to-IP Data Reuse

This section explores performance optimization opportunities that exist in current designs and whether existing solutions can exploit that.

A. Data Reuse and Reuse Distance

In a flow, data get read, processed (by IPs) and written back. The producer and consumer of the data could be two different IPs or sometimes even the same IP. We capture this IP-to-IP reuse in Figure 8, where we plotted the physical addresses accessed by the core and other IPs for `YouTube` application. Note that this figure only captures a very small slice of the entire application run. Here, we can see that the display-controller (DC) (red points) reads a captured frame from a memory region that was previously written to by video decoder (black points). Similarly, we can also see that the sound-engine reads from an address region where audio-decoder writes. This clearly shows that the data gets reused repeatedly across IPs, but the reuse distances can be very high. As mentioned in Section II-B, when a particular application is run, the same physical memory regions get used (over time) by an IP for writing different frames. In our current context, the reuse we mention is only between the producer and consumer IPs for a particular frame and nothing to to do with frames being rewritten to the same addresses. Due to frame rate requirements, reuse distances between display frame based IPs were more than tens of milli-seconds, while audio frame based IPs were less than a milli-second. Thus, there is a large variation across producer-consumer reuse distances across IPs that process large (display) frames (e.g., VD, CAM) and IPs that process smaller (audio) frames (e.g., AD, AE).

B. Converting Data Reuse into Locality

Given the data reuse, the simplest solution is to place a on-chip cache and allow the multiple IPs to share it. The

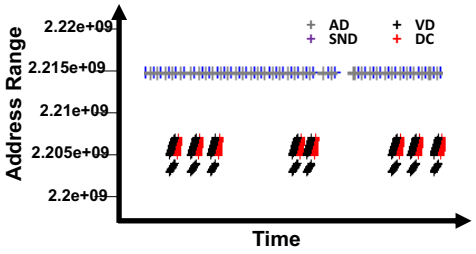

Figure 8: Data access pattern of IPs in YouTube application.

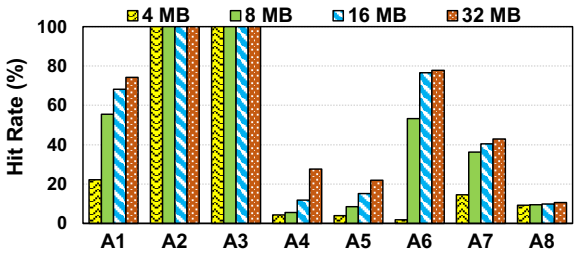

Figure 9: Hit rates under various cache capacities.

expectancy is that caches are best for locality and hence they should work. In this subsection, we evaluate the impact of adding such a shared cache to hold the data frames. Typical to conventional caches, on a cache-miss, the request is sent to the transaction queue. The shared cache is implemented as a direct-mapped structure, with multiple read and write ports, and multiple banks (with a bank size of 4MB), and the read/write/lookup latencies are modeled using CACTI [40]. We evaluated multiple cache sizes, ranging from 4MB to 32MB, and analyzed their hit rates and the reduction in cycles taken per frame to be displayed. We present the results for 4MB, 8MB, 16MB and 32MB shared caches in Figure 9 and Figure 10 for clarity. They capture the overall trend observed in our experiments. In our first experiment, we notice that as the cache sizes increase, the cache hit rates either increase or remain the same. For applications like `Audio Record` and `Audio Play` (with small frames), we notice 100% cache hit rates from 4MB cache. For other applications like `Angry Birds` or `Video-play` (with larger frames), a smaller cache does not suffice. Thus, as we increase the cache capacity, we achieve higher hit rates. Interestingly, some applications have very low cache hit rates even with large caches. This can be attributed to two main reasons. First, frame sizes are very large to fit even two frames in a large 32MB cache (as in the case of `YouTube` and `Gallery`). Second, and most importantly, if the reuse distances are large, data gets kicked out of caches by the other flows in the system or by other frames in the same flow. Applications with large reuse distances like `Video-record` exhibit such behavior.

In our second experiment, we quantify the performance benefits of having such large shared caches between IPs, and give the average cycles consumed by an IP to process a full-frame (audio/video/camera frame). As can be seen

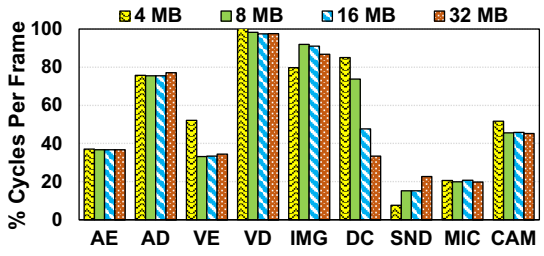

Figure 10: Cycles Per Frame under various cache capacities.

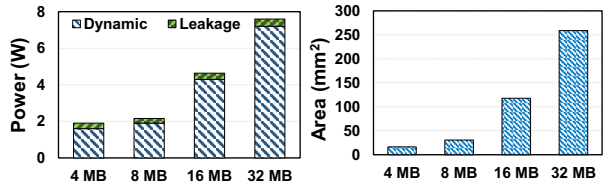

Figure 11: Area and power-overhead with large shared caches.

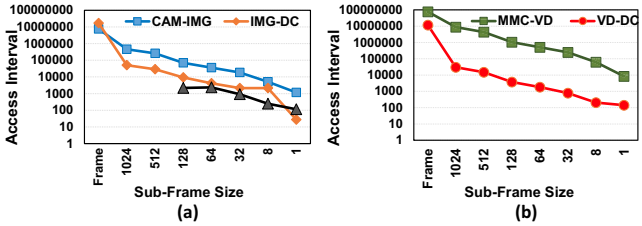

Figure 12: IP-to-IP reuse distance variation with different sub-frame sizes. Note that the y-axis is in the log scale.

from Figure 10, increasing the cache sizes does not always help and there is no optimal size. For IPs like SND and AD, the frame sizes are small and hence a smaller cache suffices. From there on, increasing cache size increases lookup latencies, and affects the access times. In other cases, like DC, as the frame sizes are large, we observe fewer cycles per frame as we increase the cache size. For other accelerators with latency tolerance, once their data fits in the cache, they encounter no performance impact.

Further, scaling cache sizes above 4MB is not reasonable due to their area and power overheads. Figure 11 plots the overheads for different cache sizes. Typically, handhelds operate in the range of 2W – 10W, which includes everything on the device (SoC+display+network). Even the 2W consumed by the 4MB cache will impact battery life severely.

Summary: To summarize, the high number of memory stalls is the main reason for frame drops, and large IP-to-IP reuse distances is the main cause for large memory stalls. Even large caches are not sufficient to capture the data reuse and hence, accelerators and devices still have considerable memory stalls. All of these observations led us to re-architect how data gets exchanged between different IPs, paving way for better performance.

V. Sub-Framing

Our primary goal is to reduce the IP-to-IP data reuse distances, and thereby reduce data stalls, which we saw were major impediment to performance in Section III.

To achieve this, we propose a novel approach of *sub-framing* the data. One of the commonly used compiler techniques to reduce the data reuse distance in loop nests that manipulate array data is to employ *loop tiling* [28], [43]. It is the process of partitioning a loop's iteration space into smaller blocks (tiles) in a manner that the data used by the loop remains in the cache enabling quicker reuse. Inspired by tiling, we propose to break the data frames into smaller *sub-frames*, that reduces IP-to-IP data reuse distances.

In current systems, IPs receive a request to process a data frame (it could be a video frame, audio frame, display frame or image frame). Once it completes its processing, the next IP in the pipeline is triggered, which in-turn triggers the following IP once it completes its processing and so on. In our solution, we propose to sub-divide these data frames into smaller sub-frames, so that once IP1 finishes it's first sub-frame, IP2 is invoked to process it. In the following sections, we show that this design reduces the hardware requirements to store and move the data considerably thereby bringing both performance and power gains. The granularity of the subframe can have a profound impact on various metrics. To quantify the effects of subdividing a frame, we varied the sub-frame sizes from 1 cache line to the current data frame size, and analyzed the reuse distances. Figure 12 plots the reduction in the IP-to-IP reuse distances (on y-axis, plotted on *log-scale*), as we reduced the size of a sub-frame. We can see from this plot an inverse exponential decrease in reuse distances. In fact, for very small sub-frame sizes, we see reuse distances in less than 100 cycles. To capitalize on such small reuse distances, we explore two techniques – *flow-buffering* and opportunistic IP-to-IP *request short-circuiting*.

A. Flow-Buffering

In Section IV-B, we showed that even large caches were not very effective in avoiding misses. This is primarily due to very large reuse distances that are present between the data-frame write by a producer and the data-frame read by a consumer. With sub-frames, the reuse distances reduce dramatically. Motivated by this, we now re-explore the option of caching data. Interestingly, in this scenario, caches of much smaller size can be far more effective (low misses). The reuse distances resulting from sub-framing are so small that even having a structure with few cache-lines is sufficient to capture the temporal locality offered by IP pipelining in SoCs. We call these structures as *flow-buffers*. Unlike a shared cache, the flow-buffers are private between any two IPs. This design avoids the conflict misses seen in a shared cache (fully associative has high power implications). These flow-buffers are write-through. As the sub-frame gets written, the sub-frame is written to memory. The reason for this design choice is discussed next.

In a typical use-case involving data flow from IP-A→IP-B→IP-C, IP-A gets its data from the main-memory and starts computing it. During this process, as it completes a sub-frame, it writes back this chunk of data into the flow-buffer between IP-A and IP-B. IP-B starts processing this sub-frame from the flow-buffer (in parallel with IP-A working on another sub-frame) and writes it back to the flow-buffer between itself and IP-C. Once IP-C is done, the data is written into the memory or the display. Originally, every read and write in the above scenario would have been scheduled to reach the main memory. Now, with the flow-buffers in place, all the requests can be serviced from these

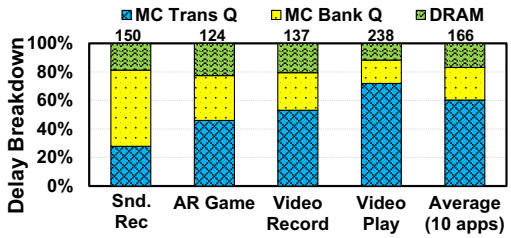

Figure 13: Delay breakdown of a memory request issued by IPs or cores. The numbers above the bar give the absolute cycles.

small low-latency cost and area efficient buffers.

Note that, in these use-cases, cores typically run device driver code and handle interrupts. They have minimal data frames processing. Consequently, we do not incorporate flow-buffers between core and any other accelerator. Also, when a use-case is in its steady-state (for example, a minute into running a video), the IPs are in the active state and quickly consume data. However, if an IP is finishing up on an activity or busy with another activity or waking up from sleep state, the sub-frames can be overwritten in the flow-buffer. In that case, based on sub-frame addresses, the consumer IP can find its data in the main memory since the flow-buffer is a write-through buffer. In our experiments, discussed later in Section VII, we found that a flow-buffer size of 32 KB provides a good trade-off between avoiding a large flow-buffer and sub-frames getting overwritten.

B. IP-IP Short-circuiting

The flow-buffer solution requires an extra piece of hardware to work. To avoid the cost of adding the flow-buffers, an alternate technique would be to enable consumers directly use the data that their producers provide. Towards that, we analyzed the average round-trip delays of all accesses issued by the cores or IPs (shown in Figure 13) and found requests spend maximum time queuing in the memory subsystem. `MC Trans Queue` shows the time taken from the request leaving the IP till it gets to the head of the transaction queue. The next part `MC Bank Queue`, is the time spent in bank queues. This is primarily determined by whether the data access was a row buffer hit, or miss. And, finally `DRAM` shows the time for DRAM accessing along with the response back to the IPs. As can be seen, most of the time is spent in the memory transaction queues (~100 cycles). This means that data that could otherwise be reused lies idle in the memory queues and we use this observation towards building an opportunistic IP-to-IP short-circuiting technique, similar in concept to "store-load forwarding" in CPU cores [29], [38][2] though our technique is in between different IPs. There are correctness and implementation differences, which we highlighted in the following paragraphs/sections.

[2]Core requests spend relatively insignificant amount of time in transaction queues as they are not bursty in nature. Due to their strict QoS deadlines, they are prioritized over other IP requests. They spend more time in bank queues and in DRAM.

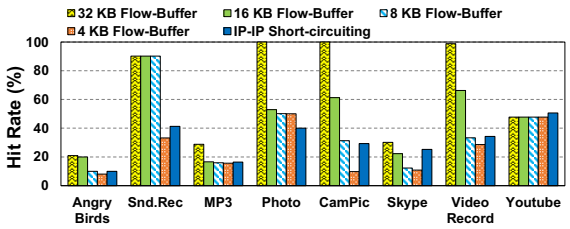

Figure 14: Hit rates with flow-buffering and IP-IP short-circuiting.

IPs usually load the data frames produced by other IPs. Similar to store-load forwarding, if the consumer IP's load requests can be satisfied from the memory transaction queue or bank queues, the memory stall time can be considerably reduced. As the sub-frame size gets smaller, the probability of a load hitting a store gets higher. Unlike the flow-buffers discussed in Section V-A, store data does not remain in the queues till they are overwritten. This technique is opportunistic and as the memory bank clears up its entries, the request moves from the transaction queue into the bank queues and eventually into main memory. Thus, the loads need to follow the stores quickly, else it has to go to memory. This distance between the consumer IP load request and producer IP store request depends on how full the transaction and bank queues are. In the extreme case, if both the queues (transaction-queue and bank-queue) are full, the number of requests that a load can come after a store will be the sum of the number of entries in the queues.

The overhead of implementing the IP-IP short-circuiting is not significant since we are using pre-existing queues present in the system agent. The transaction and bank queues already implement an associative search to re-order requests based on their QoS requirements and row-buffer hits, respectively [33]. Address-searches for satisfying core loads already exist and these can be reused for other IPs. As we will show later, this technique works only when the sub-frame reuse distance is small.

C. Effects of Sub-framing Data with Flow-Buffering and IP-IP Short-circuiting

The benefits of sub-framing are quantified in Figure 14 in terms of hit rates when using flow-buffering and IP-IP short-circuiting. We can see that the buffer hit rates increase as we increase the size of flow-buffers, and saturate when the size of buffers are in the ranges of 16KB to 32KB. The other advantage of having sub-frames is the reduced bandwidth consumption due to the reduced number of memory accesses. As discussed before, accelerators primarily face bandwidth issues with the current memory subsystem. Sub-framing alleviates such bottleneck by avoiding fetching every piece of data from memory. Redundant writes and reads to same addresses are avoided. Latency benefits of our techniques, as well as their impact on user experience will be given later in Section VII.

VI. Implementation Details

In implementing our sub-frame idea, we account for the probable presence of dependencies and correctness issues resulting from splitting frames. Below, we discuss the correctness issue and the associated intricacies that need to be addressed to implement sub-frames. We then discuss the software, hardware and system-level support needed for such implementations.

A. Correctness

We broadly categorize data frames into the following types – (i) video, (ii) audio, (iii) graphics display, and (iv) the network packets. Of these, the first three types of frames are the ones that usually demand sustained high bandwidth with the frame sizes varying from a megabyte to tens of MBs. In this work, we address only the first three types of frames, and leave out network packets as the latency of network packet transmission is considerably higher compared to the time spent in the SoC.

Video and Audio Frames: Encoding and decoding, abbreviated as *codec* is compression and decompression of data that can be performed at either hardware or software layer. Current generation of smartphones such as Samsung S5 [36] and Apple iPhone [3] have multiple types of codes embedded in their phone.

Video Codecs: First, let us consider the flows containing video frames, and analyze the correctness of sub-dividing such large frames into smaller ones. Among the video codecs, the most commonly used are H.264 (MPEG-4) or H.265 (MPEG-H, HEVC) codecs. Let us take a small set of video frames and analyze the decoding process. The encoding process is almost equivalent to the inversion of each stage of decoding. As a result, similar principles apply there as well. Figure 15 shows a video clip in its entirety, with each frame component named. A high-quality HD video is made up of multiple frames of data. Assuming a default of 60 FPS, the amount of data needed to show the clip for a minute would be 1920x1080(screen resolution) x 3(bytes/pixel) x 60 (frame rate) x 60 (seconds) = 21.35 GB. Even if each frame is compressed individually and stored on today's hand-held devices, the amount of storage available would not permit it. To overcome this limitation, codecs take advantage of the temporal redundancy present in video frames, as the next frame is usually not very different from the previous frame.

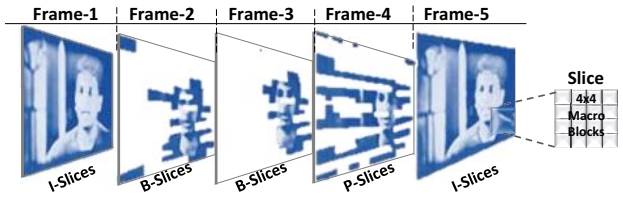

Figure 15: Pictorial representation showing the structure of five consecutive video frames.

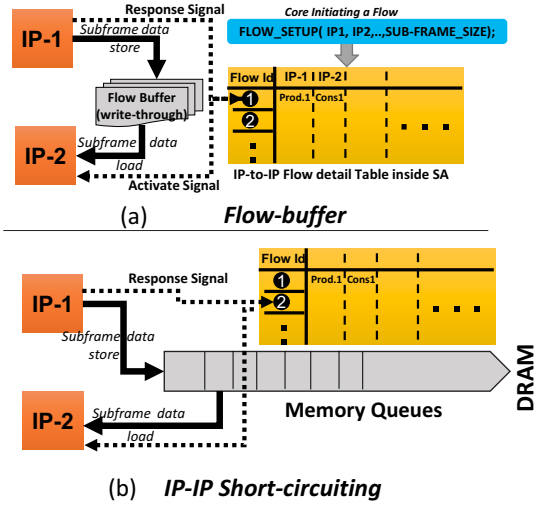

Figure 16: High level view of the SA that handles sub-frames.

Each frame can be dissected into multiple slices. Slices are further split into macroblocks, which are usually a block of 16x16 pixels. These macroblocks can be further divided into finer granularities such as sub-macroblocks or pixel blocks. But, we do not need such fine granularities for our purpose. Slices can be classified into 3 major types: I-Slice (independent or intra slices), P-Slice (predictive), and B-Slice (bi-directional) [37] as depicted in Figure 15. [3] I-slices have all data contained in them, and do not need motion prediction. P-slices use motion prediction from one slice which belongs to the past or future. B-slices use two slices from past or the future. Each slice is an independent entity and can be decoded without the need for any other slice in the *current frame*. P- and B-slices need slices from a previous or next frame only.

In our sub-frame implementation, we choose *slice-level granularity* as the finest level of sub-division to ensure correctness *without having any extra overhead of synchronization*. As slices are independently decoded in a frame, the need for another slice in the frame does not arise, and we can be sure that correctness is maintained. Sub-dividing any further would bring in dependencies, stale data and overwrites.

Audio Codecs: Audio data is coded in a much simpler fashion than video data. An audio file has a large number of frames, with each audio frame having the same number of bits. Each frame is independent of another and it consist of a header block and data block. Header block (in MP3 format) stores 32-bits of metadata about the coming data block frame. Thus, each audio frame can be decoded independently of another as all required data for decoding is present in the current frames header. Therefore, using a full audio frame as a sub-frame would not cause any correctness issue.

Graphics Rendering: Graphics IPs already employ tiled

[3] Earlier codecs had frame level classification instead of slice level. In such situations, I-frame is constructed as a frame with only I-slices.

rendering when operating on frames and for the display rendering. These tiles are similar to the sub-frames proposed in this work. A typical tiled rendering algorithm first transforms the geometries to be drawn (multiple objects on the screen) into screen space and assigns them into tiles. Each screen-space tile holds a list of geometries that needs to be drawn in that tile. This list is sorted front to back, and the geometry behind another is clipped away and only the ones in the front are drawn to the screen. GPUs renders each tile separately to a local on-chip buffer, which is finally written back to main-memory inside a framebuffer region. From this region, the display controller reads the frame to be displayed to be shown on screen. All tiles are independent of each other, and thus form a sub-frame in our system.

B. OS and Hardware Support

In current systems, IPs are invoked sequentially one-after-another per frame. Let us revisit the example considered previously – a flow with 3 IPs. The OS, through device drivers, calls the first IP in the flow. It waits for the processing to complete and the data to be written back to memory and then calls the second IP. After the second IP finishes its processing, the third IP is called. With sub-frames, when the data is ready in the first IP, the second IP is notified of the incoming request so that it can be ready (by entering to the active state from a low power state) when the data arrives. We envision that the OS can capture this information through a library (or multiple libraries) since the IP-flows for each application are pretty standard. In Android [15] for instance, there is a layer of libraries (Hardware Abstraction Layer – HAL) that interface with the underlying device drivers and these HALs are specific to IPs. As devices evolve, HAL and the corresponding drivers are expected to enable access to devices to run different applications. By adding an SA HAL and its driver counterpart to communicate the flow information, we can accomplish our requirements. From the application's perspective, this is transparent since the access to the SA HAL happens from within other HALs as they are requested by the applications. Figure 16 shows a high level view of the sub-frame implementation in SA along with our short-circuiting techniques. From a hardware perspective, to enable sub-framing of data, the SA needs to have a small matrix of all IPs – rows corresponding to producers and columns to consumers. Each entry in the row is 1 bit per IP. Currently, we are looking at about 8 IPs, and this is about 8 bytes in total. In future, even as we grow to 100 IPs, the size of the matrix is small. As each IP completes its sub-frame, the SA looks at its matrix and informs the consumer IP. In situations where we have multiple flows (currently Android allows two applications to run simultaneously [32]) with an IP in common, the entries in the SA for the common IP can be swapped in or out along with the context of the application running. This will maintain the correct consumer IP status in the matrix for any IP.

VII. Evaluation

In this section, we present the performance and power benefits obtained by using sub-frames compared to the conventional baseline system which uses full frames in IP flows. We used a modified version of the GemDroid infrastructure [7] for the evaluation. For each application evaluated, we captured and ran the traces either to completion or for a fixed time. The trace lengths varied from about 2 secs to 30 secs. Usually this length is limited by frame sizes we needed to capture. Workloads evaluated are listed in Table II and the configuration of the system used is described in Section II-D and Table III. For this evaluation we used a sub-frame size of 32 cache lines (2KB). The transaction queue and bank queues in SA can hold 64 entries (totaling 8KB). For the flow buffer solution, we used a 32 KB buffer (based on the hit rates observed in Figure 14).

User Experience: As a measure of user experience, we track the number of frames that could complete in each flow. The more frames that get completed, lesser the frame drops and better is the user experience. Figure 17 shows the number of frames completed in different schemes. The y-axis shows the percentage of frames completed out of the total frames in an application trace. The first bar shows the frames completed in the baseline system with full frame flows. The second and third bars show the percentage of frames completed with our two techniques. In baseline system, only around 76% of frames were displayed. By using our two techniques, the percentage of frames completed improved to 92% and 88%, respectively. Improvements in our schemes are mainly attributed to the reduced memory bandwidth demand and improved memory latency as the consumer IP's requests are served through the flow-buffers or by short-circuiting the memory requests. The hit-rates of consumer's requests were previously shown in Figure 14. In some cases, flow-buffers perform better than short-circuiting due to the space advantage in the flow buffering technique.

Performance Gains: To understand the effectiveness of our schemes, we plot the average number of cycles taken to process a frame in each flow in Figure 18. This is the time between the invocation of first IP and completion of last IP in each flow. Note that, reducing the cycles per frame can lead to fewer frame drops. When we use our techniques with sub-framing, due to pipelining of intra-frame data across multiple IPs instead of sequentially processing one frame after another, we are able to substantially reduce the cycles per frame by 45% on average. We also observed that in A6-Skype application (which has multiple flows), through the use of sub-framing, the memory subsystem gets overwhelmed because, we allow more IPs to work at the same time. This is not the case in the base system. If IPs do not benefit from flow-buffers or IP request short-circuiting, the memory pressure is more than the baseline leading to some performance loss (17%).

Energy Gains: Energy efficiency is a very important

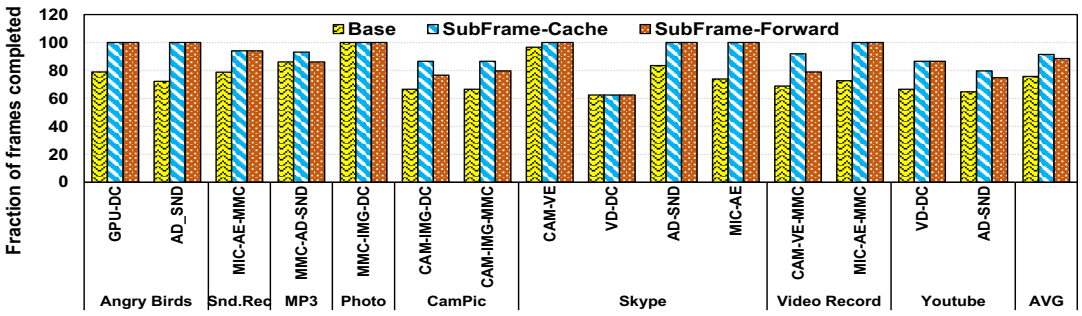

Figure 17: Percentage of Frames Completed (Higher the better).

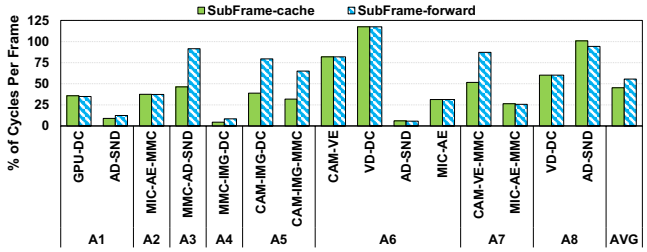

Figure 18: Reduction in Cycles Per Frame in a flow normalized to Baseline (Lower the better).

metric in handhelds since they operate out of a battery (most of the time). Exact IP design and power states incorporated are not available publicly. As a proxy, we use the number of cycles an IP was active to correspond to the energy consumed when running the specific applications. In Figure 19, we plot the total number of active cycles consumed by an accelerator compared to the base case. We plot this graph only for accelerators as they are compute-intensive and hence, consume most of the power in a flow. On average, we observe 46% and 35% reduction in active cycles (going up to 80% in GPU) with our techniques, which translates to substantial system-level energy gains. With sub-framing, we also reduce the memory energy consumption by 33% due to (1) reduced DRAM accesses, and (2) memory spending more time in low-power mode. From the above results it can be concluded that sub-framing yields significant performance and power gains.

VIII. Related Work

Data Reuse: Data reuse within and across cores has been studied by many works. Chen et al. [6], [47], Gordon et al. [17] and Kandemir et al. [22] propose compiler optimizations that perform code restructuring and enable data sharing

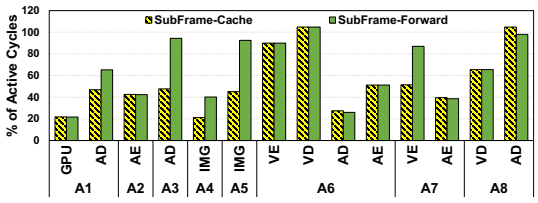

Figure 19: Reduction in Number of Active Cycles of Accelerators (Lower the better).

across processors. Suhendra et al., [45] proposed ways to optimally use scratch pad memory in MPSoCs along with methods to schedule processes to cores. There have been multiple works that discuss application and task mapping to MPSoCs [9], [30], [41] with the goal of minimizing data movement across cores. Our work looks at accelerator traffic, which is dominant in SoCs, and identifies that frame data is reused across IPs. Unlike core traffic, the reuse can be exploited only if the data frames are broken in sub-frames. We capture this for data frames of different classes of applications (audio/video/graphics) and propose techniques to reduce the data movement by short circuiting the producer writes to the consumer reads.

Memory Controller Design: A large body of works exist in the area of memory scheduling techniques and memory controller designs in the context of MPSoCs. Lee and Chang [27] describe the essential issues in memory system design for SoCs. Akesson et al. [2] propose a memory scheduling technique that provides a guaranteed minimum bandwidth and maximum latency bound to IPs. Jeong et al. [20] provide QoS guarantees to frames by balancing memory requests at the memory controller. Our work identifies a specific characteristic (reuse at sub-frame level) that exists when data flows through accelerators and optimizes system agent design. Our solution is complimentary to prior techniques and can work in tandem with them.

Along with IP design and analysis, several works have proposed IP-specific optimizations [14], [19], [24], [39] and low power aspects of system-on-chip architectures [11], [18], [46]. Our solution is not specific to any IP rather, it is at the system-level. By reducing the IP stall times and memory traffic, we make the SoC performance and power-efficient.

IX. Conclusions

Memory traffic is a performance bottleneck in mobile systems, and it is critical that we optimize the system as a whole to avoid such bottlenecks. Media and graphics applications are very popular on mobile devices, and operate on data frames. These applications have a lot of temporal locality of data between producer and consumer IPs. We show that by operating a frame as an atomic block between different IPs, the reuse distances are very high, and thus, the

available locality across IPs goes unexploited. By breaking the data frames into sub-frames, the reuse distance decreases substantially, and we can use flow buffers or the existing memory controller queues to forward data from the producer to consumer. Our results show that such techniques help to reduce frame latencies, which in turn enhance the end-user experience while minimizing energy consumption.

ACKNOWLEDGMENTS

This research is supported in part by the following NSF grants – #1205618, #1213052, #1302225, #1302557, #1317560, #1320478, #1409095, #1439021, #1439057, and grants from Intel.

REFERENCES

[1] "Vision Statement: How People Really Use Mobile," January-February 2013. [Online]. Available: http://hbr.org/2013/01/how-people-really-use-mobile/ar/1

[2] B. Akesson *et al.*, "Predator: A predictable sdram memory controller," in *CODES+ISSS*, 2007.

[3] Apple, "Apple iphone 5s." Available: https://www.apple.com/iphone/

[4] N. Balasubramanian *et al.*, "Energy consumption in mobile phones: A measurement study and implications for network applications," in *IMC*, 2009.

[5] M. N. Bojnordi and E. Ipek, "Pardis: A programmable memory controller for the ddrx interfacing standards," in *ISCA*, 2012.

[6] G. Chen *et al.*, "Exploiting inter-processor data sharing for improving behavior of multi-processor socs," in *ISVLSI*, 2005.

[7] N. Chidambaram Nachiappan *et al.*, "GemDroid: A Framework to Evaluate Mobile Platforms," in *SIGMETRICS*, 2014.

[8] P. Conway *et al.*, "Cache hierarchy and memory subsystem of the amd opteron processor," *IEEE micro*, 2010.

[9] A. Coskun *et al.*, "Temperature aware task scheduling in mpsocs," in *DATE*, 2007.

[10] R. Das *et al.*, "Aergia: Exploiting packet latency slack in on-chip networks," in *ISCA*, 2010.

[11] B. Diniz *et al.*, "Limiting the power consumption of main memory." in *ISCA*, 2007.

[12] J. Engwell, "The high resolution future âĂŞ retina displays and design," Blurgroup, Tech. Rep., 2013.

[13] J. Y. C. Engwell, "Gpu technology trends and future requirements," Nvidia Corp., Tech. Rep.

[14] S. Fenney, "Texture compression using low-frequency signal modulation," in *HWWS*, 2003.

[15] Google, "Android HAL." Available: https://source.android.com/devices/index.html

[16] Google. (2013) Android sdk - emulator. Available: http://developer.android.com/

[17] M. I. Gordon *et al.*, "A stream compiler for communication-exposed architectures," in *ASPLOS*, 2002.

[18] A. Gutierrez *et al.*, "Full-system analysis and characterization of interactive smartphone applications," in *IISWC*, 2011.

[19] K. Han *et al.*, "A hybrid display frame buffer architecture for energy efficient display subsystems," in *ISLPED*, 2013.

[20] M. K. Jeong *et al.*, "A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," in *DAC*, 2012.

[21] A. Jog *et al.*, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

[22] M. Kandemir *et al.*, "Exploiting shared scratch pad memory space in embedded multiprocessor systems," in *DAC*, 2002.

[23] C. N. Keltcher *et al.*, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, 2003.

[24] H. b. T. Khan and M. K. Anwar, "Quality-aware Frame Skipping for MPEG-2 Video Based on Inter-frame Similarity," Malardalen University, Tech. Rep.

[25] Y. Kim *et al.*, "ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.

[26] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[27] K.-B. Lee and T.-S. Chang, *Essential Issues in SoC Design Designing - Complex Systems-on-Chip*. Springer, 2006, ch. SoC Memory System Design.

[28] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *POPL*, 1997.

[29] G. H. Loh *et al.*, "Memory bypassing: Not worth the effort," in *WDDD*, 2002.

[30] P. Marwedel *et al.*, "Mapping of applications to mpsocs," in *CODES+ISSS*, 2011.

[31] A. Naveh *et al.*, "Power management architecture of the 2nd generation intel® coreâĎć microarchitecture, formerly codenamed sandy bridge," 2011.

[32] S. T. Report, "Wqxga solution with exynos dual," 2012.

[33] S. Rixner *et al.*, "Memory access scheduling," in *ISCA*, 2000.

[34] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator," *CAL*, 2011.

[35] R. Saleh *et al.*, "System-on-chip: Reuse and integration," *Proceedings of the IEEE*, 2006.

[36] Samsung, "Samsung galaxy s5," 2014. Available: http://www.samsung.com/global/microsite/galaxys5/

[37] H. Schwarz *et al.*, "Overview of the scalable video coding extension of the h. 264/avc standard," *IEEE Transactions on Circuits and Systems for Video Technology*, 2007.

[38] T. Sha *et al.*, "Scalable store-load forwarding via store queue index prediction," in *MICRO*, 2005.

[39] H. Shim *et al.*, "A compressed frame buffer to reduce display power consumption in mobile systems," in *ASP-DAC*, 2004.

[40] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," Technical Report 2001/2, Compaq Computer Corporation, Tech. Rep., 2001.

[41] A. K. Singh *et al.*, "Communication-aware heuristics for run-time task mapping on noc-based mpsoc platforms," *J. Syst. Archit.*, 2010.

[42] R. SoC, "Rk3188 multimedia codec benchmark," 2011.

[43] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," in *ACM SIGPLAN Notices*, 1999.

[44] D. S. Steve Scheirey, "Sensor fusion, sensor hubs and the future of smartphone intelligence," ARM, Tech. Rep., 2013.

[45] V. Suhendra *et al.*, "Integrated scratchpad memory optimization and task scheduling for mpsoc architectures," in *CASES*, 2006.

[46] Y. Wang *et al.*, "Markov-optimal sensing policy for user state estimation in mobile devices," in *IPSN*, 2010.

[47] L. Xue *et al.*, "Spm conscious loop scheduling for embedded chip multiprocessors," in *ICPADS*, 2006.

[48] M. Yuffe *et al.*, "A fully integrated multi-cpu, gpu and memory controller 32nm processor," in *ISSCC*, 2011.

[49] Y. Zhang *et al.*, "A data layout optimization framework for nuca-based multicores," in *MICRO*, 2011.

[50] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *HPCA*, 2013.